DISEÑAR E IMPLEMENTAR UNA APLICACIÓN DE PREPROCESADO DE CÓDIGO C/C++ BASADO EN CLANG.

Grado en Ingeniería Informática

Curso: 2017 - 2018

Autor: Alejandro Martínez-Valero López-Manterola

Tutor: Alejandro Calderón Mateos

Universidad Carlos III de Madrid

Índice de contenido

Índice de ilustraciones

Índice de tablas

Abstract

The aim of this section is to summarize the content on the document, providing the most remarkable info.

In order to for it to be clear, its contents will be organized to follow the same section structure as the rest of the document.

1. Introduction

In this chapter the aim is to present the motivations behind this project development and the main goals that it will try to achieve.

It is also included an explanation of the document structure alongside with the details of the content of each chapter.

1.1. Motivation

Currently, in the Information Age, software development companies have a great impact on society, either because of their economic footprint or because of its influence on improving the quality of life.

An important part of the life cycle of a software product is its maintenance, and it is a tedious task, since it is done on the finished application, and a change may involve the revision of a large part of the source code. Worse yet, such a review may fall on a worker oblivious to the original implementation of the code, either because the original author no longer works at the company or because he is busy with another task.

Many times this task is not difficult, but long, repetitive and prone to errors, which requires resources that could be used elsewhere. This is the motivation behind this end-of-degree project: to demonstrate that these tasks can be automated.

1.2. Objetives

Now that the motivation has been explained, the following objectives are presented:

- **Find the most appropriate tool to develop this application.** We must take into account the programming languages that the product will have to support.
- **Develop the application.** Since it is intended to modify source code before its compilation, the application must be able to analyze the code and understand its structure. As the range of possible utilities that benefit of this feature is very wide, it has been decided to develop a monitoring tool, which will need to find specific areas of the code and then modify them.
- **Add more features.** Once the previous objective has been achieved, it will be necessary to study what functionalities could be added to the product in a way that best suits the reasons for its development.

1.3. Document structure

Below is the structure that the document will follow, with a brief description of the content of each section.

1. **Introduction.** Contains the motivation, objectives and structure of the document.
2. **State of the art.** It contains the existing tools related to the motivation of this TFG.
3. **Analysis and design.** It contains the requirements specification and the decisions made in the design of the product.
4. **Implementation and implementation.** Contains the details of the implementation and implementation of the tool.
5. **Evaluation.** It contains the test plan with which it has been verified that the implementation of the requirements has been correct.
6. **Socio-economic environment.** It contains the planning and budget of the project and its socio-economic impact.
7. **Conclusions and future work.** It contains the conclusions obtained from the development of the project and a list of possible improvements applicable to the product.

8. **Annex I: Repository**. It contains the web address of the repository where you can find all the files generated during the course of this project and the installation instructions of the development environment.

2. State of the art

In this chapter, the current situation of automatic source to source transformation will be discussed. Then a comparative of the different approaches to the development of a tool that performs that transformation is presented.

First it will be presented that after a prolonged search, **the only tool found to make use of code analysis at the same level of what is being developed in this project is a transpiler made by another student of this university [1]**.

It is worth noting that the use of source to source transformation is not new, but it is been used less deeply of what this project aims to do. Some examples of this transformations are functionalities added to IDE like IntelliJ [2] and Eclipse [3], like refactorization, in which a variable or function can be renamed in one file and the IDE will update all calls to it in all files at the same time, or the automatic generation of get() and set() functions in Java.

Also there are lots of monitoring tools [4], but all found so far work from outside the monitored application, having limitations on which information they can get from it. Since our application will work from the inside there should be no such limitations.

Since this tool [1] can only perform Java code analysis, it was decided to focus making our application compatible with more languages at the same time. Making use of the information obtained from TIOBE website [5], it appears that the best bet is to aim to be compatible with C and C++.

Once it was decided the goal languages, it was time to choose which tool to use for developing our project. There are many choices for parser generators, as seen here [6], but not all of them are suitable for source code transformation or are compatible with C and C++. The three best choices are Bison[7] and Flex[8], ANTLR [9] or Clang [10].

Bison is a parser generator and Flex is a scanner generator. Using them together they can generate a tool capable of recognize code and act accordingly to what the user specifies. They support the C and C++ languages, but not

simultaneously. And, since the user has to implement the grammars for both tools to generate the desired application, they provide a low level of abstraction.

ANTLR is a parser generator, with a scanner incorporated. It only needs one file with both the lexer rules and the parser rules, unlike the previous tools which needed them to be in separated files. it is also compatible with C and C++ languages, and in addition, thanks to this repository [11], the grammars are already written. However, it also provides a low-level abstraction since it still uses grammars.

Clang is the front end for languages in the C family (C, C ++, Objective C / C ++, OpenCL, CUDA, and RenderScript) of the LLVM compiler [10][12].

It also provides the necessary infrastructure to develop tools that work with source code in those languages, through three interfaces [13].

Unlike the previous ones, to develop a tool it is not necessary to specify any grammar, since we work on the AST generated by Clang, so the work is at a more abstract level. Depending on the interface chosen to communicate with Clang, the user can interact with the AST with greater or lesser depth or even develop tools capable of working on any machine, even without Clang or LLVM installed. In addition, it also offers the user freedom to choose in which language he wants to develop the tool.

Since all three options for developing the project's application are compatible with C and C++ all three are valid. However, **the tool chosen at the end was Clang**, mostly because of the high level of abstraction, which will make easier the development.

3. Analysis and design

In this chapter it will be presented the result of the execution of the requisite analysis and the design decisions that were made.

The tables containing the requirement specification are in the chapter 3 of the Spanish version Análisis y Diseño.

Regarding the design choices, as mentioned before, Clang tool offers three interfaces for application development:

- LibClang: LibClang is a backward compatible interface, so a tool created with this interface would not need to be modified when updating Clang. It also allows the tool to be implemented in a language other than C ++, although it does not give full control over the AST that Clang generates.
- Clang Plugins: Plugins are dynamic libraries that are loaded at runtime by the compiler. They offer access to all Clang features, but tools made with this interface cannot be used outside the Clang development environment.
- LibTooling: It is an interface in C ++ that has access to all Clang features and can be compiled so that it can be used on any machine, even if it does not have the Clang development environment. It allows you to run the tool on one or more files manually, unlike the plugins, which can be automated to act when there is a dependency change.

The chosen interface is LibTooling, since it does not force the end user to have installed in the Clang development environment, and it has all the functionalities.

It has also been decided that the project's tool will only introduce calls to functions in the target source code. Those functions that will be found in an external library that will be provided with the tool.

Finally there are two laws that affect the application, the data protection law [14] and the intellectual protection law [15].

Since the tool does not collect any data, it will not be necessary to inform or request permission to process it to the user to be in accordance to the data protection law.

Regarding the intellectual protection law, Clang [16] uses the BSD license [17], which allows the use of the code for both free and proprietary software.

4. Implementation and deployment

The aim in this chapter is to expose the obstacles that emerged during the development phase and the description of the machine that was used.

4.1. Implementation

This section will present the obstacles found during the implementation phase of the project.

The first thing is to emphasize that despite the existence of official and recent documentation [18], it seems not to have been reviewed in detail, since some steps are not feasible, such as the installation of Cmake that uses a repository that no longer serves . It should also be noted that other steps do not work because of the lack of intermediate steps not specified.

In addition, despite the reasonable amount of RAM available to the machine where the development environment was being installed, the installation stopped frequently due to lack of memory, and it became necessary to acquire a new machine with more memory. Even so the installation failed many times, but luckily when retrying it resumed at the point where the previous installation stopped. In the end the preparation of the development environment lasted several days, mostly because of the slow and faulty installation.

After saving all these obstacles, we proceeded to compile a tool that simply showed the AST generated by Clang, and it was discovered that this process requires several minutes, since all the additional code necessary for the tool to work as a standalone application is being compiled. Although understandable, this delayed mildly the implementation because, although the changes were accumulated to reduce the number of compilations, there were still many compilations.

The development was divided into several prototypes, each one of greater complexity than the previous one, while studying the different solutions to the proposed problems that arisen during their implementation.

The first prototype aimed to verify the correct installation of the development environment and the ability of Clang to generate an AST. With its implementation, it was learned about the three main classes that will comprise the backbone of the final application, the ASTFrontendAction, the ASTConsumer and the ASTRecursiveVisitor.

It should be noted that the official tutorials do not go much further than this point. While it is true that there is documentation of all the classes that make up Clang, normally it does not detail anything more than the relations of one class with the others and the methods available to it, so it has had to make extensive use of engines Search and Stack Overflow.

The objective of the second prototype was to detect the start and end of all the function declarations, and that of the if statements.

In this prototype an obstacle was found: Clang offers two functions, getLocStart() and getLocEnd() to determine the start and end of the zone in which you are working. The first works perfectly, but the second one always points a token ahead of where it would be expected. This means that any rewriting that used it as is, would end up inside the last expression in the work area instead of after it. It seems to be a bug that Clang has been dragging for some time, but in the end it was possible to solve the issue that it generated.

With this second prototype, it was learned about the Rewriter class and how it works, including the managing of the working scope and the output of the modified code.

The third prototype was designed to find calls to a specific function - in this case the malloc function - and specify what function called it. Here we find the following obstacle: finding the tree nodes corresponding to calls to malloc is easy, but Clang does not give us any way to get the caller from them. The first solution proposed was to use another class to find the node with a regular expression, but this class does not have access to the Rewriter class. This caused the solution to be discarded and use another one. The solution at the end lies in the order in which Clang travels the nodes: both the caller and the call are nodes of type FunctionDecl, but the caller is always traversed first and, unlike the callee, it has a function with body so that a simple call to the hasBody() function and the use of a global variable allows us to obtain both.

With all the knowledge gathered from the development of the three previous prototypes, the implementation of the project's application went through without any more complications.

4.2. Deployment

Since the tool has been compiled so that all the code it will need is included in it, the only requirement to use this tool is to use a machine with an operating system compatible with the Debian family.

Anyway, in case there is a problem, the development environment has been Clang 8 on a Linux Mint 18.1 operating system on an HP Notebook 250 G6 Intel Core i5-7200U notebook with 8GB of RAM, 256GB of solid state hard disk and a 15.6 "screen.

The minimum requirements for developing this tool are this:

- **60 GB of free disk space.** The development environment requires the download and build of the LLVM and Clang tools. The downloaded files occupy less than 5GB, but the build files folder increases its weigh rapidly during the installation process.
- **8 GB of RAM memory.** Using less than this amount will increase installation times and failures way over reason.

The processor speed does not seem to affect installation time to much since the real bottleneck is in the RAM memory.

5. Evaluation

In this section, it is explained the process followed to test the project's application.

First a series tests were designed so every requirement obtained during the requirement analysis phase of the project is verified at least once.

After testing the application with each test generated, **the application proved having met all the requirements of the project.**

6. Socio-economic environment

In this section, it is explained the planning, the budget and the impact the application will have in society.

6.1. Planning

After logging all the hours spent in the development of this project, it has been observed some delays that have affected the estimated completion date of the project. It should be noted that the most notable delay was in the preparation of the development environment, as the official documentation did not specify the minimum system requirements for the installation. Therefore, after several days trying to figure out the reason the installation was been so slow and faulty, it was determined the necessity to obtain a new machine to install the environment in and, despite the improvement in the amount of RAM and the use of an SSD, it still took several days more than planned to start developing the tool.

There were also delays during the implementation due to the lack of documentation, which made it very difficult to resolve the unexpected obstacles

that were encountered during the prototype development as the prototype's functionalities increased.

6.2. Budget

After the planning a budget was made to obtain the possible costs if a company decides to develop the project. It has to be noted that **the costs of things like the building rent and water and electricity bills were not included in the calculations of the budget.**

To find more specific details on the actual numbers or their sources please refer to chapter 6.2. Presupuesto

Since the real planning took more time than the estimations, the excess cost is to be subtracted from the risk margin and, in the case it were necessary, from the benefit margin. **However the delay is well covered in the risk margin,** so the benefits obtained from the development of the project are untouched.

6.3. Socio-economic impact

As already mentioned in the motivation section, companies are forced to spend a significant amount of resources on the maintenance of their software products.

This end-of-degree project is a demonstration of the ability to develop tools that perform such maintenance automatically today.

Each task that is automated will free resources that can be used to carry out other lucrative projects for the company.

Another result of this automation is that, having more resources, companies take advantage to improve the service provided by their products, which will have a positive effect on customers and users.

For this reason it is observed that **the development of applications with the aim of automating the maintenance of software products will have a positive impact, both economically for the companies, and in the quality of life.**

7. Conclusions and future works

Enclosed in the following section are the conclusions raised from developing this project and the possible functionalities that can be added to it.

7.1. Conclusions

Regarding the product, it is worth highlighting the number of possibilities offered by the automatic code modification, and its power. A very common example today, so much that it is difficult to think that at some point this functionality did not exist, it is refactoring.

With Clang all these possibilities are within reach. Since it generates an abstract syntactic tree, you can implement the changes you want with very little code. And, when dealing with data from a more abstract level than other tools, it is transparent to the user.

This project has demonstrated the ease of developing applications in this field using Clang: with very few lines of code you can get very powerful tools. For example, the first prototype implemented has 58 lines, the second has 156 and the third 180. It should be noted that the third prototype has all the features of the previous prototype and that the final tool does not have many more lines.

In fact, once you have learned the general behavior of applications that use the LibTooling interface, it does not take much time to create them from scratch or to reform existing code.

Personally this end-of-degree project has been very rewarding, since the author has always found the work of compilers fascinating, and their ability to understand and interpret the source code very interesting. Since the tool proposed in this project does not change the level of abstraction of the language, in reality it is a transpiler, but all transpilers are compilers at the end of the day.

Finally, the development of this project has helped to reinforce the author's knowledge especially in C ++, since it has not been a very requested language during the development of his career.

7.2. Future Works

Below are some ideas that could be incorporated into the tool developed in this project:

- **Selection of the functions to be monitored.** Currently, if the monitoring functionality is specified, all the functions found in the source code are

monitored. This is not always desirable and allowing such selection would be an improvement.

- **Refactor.** Although this can easily be done in a modern editor, adding this functionality would also be beneficial.
- **Selection of the name of the output file.** In the current state of the product, all generated files have the same name as their original with the addition of an "M" at the end. Therefore, it would be good to allow the user to decide the name of the output file.
- **Use of a configuration file.** As more features are added with their corresponding options, the complexity of the execution command is increased, making it prone to errors. That is why it is proposed to do as Maven, and move all these configuration options to a file, which would be read when the tool is executed.
- **Integration with IDE.** Making the tool executable from an IDE makes it more attractive and easy to use, especially if the previous point is implemented.

It is also worth mentioning a completely different product: a tool that, using artificial intelligence, learns the coding style of a programmer, and with that information transforms the code that it writes so that it conforms to the standards of the company in which it works. It would also work in reverse, take the code from the company repository and rewrite it so that the programmer is comfortable.

1. Introducción

En este apartado se explicará las motivaciones y objetivos del presente trabajo de fin de grado.

También se describirá brevemente el contenido de cada apartado del documento.

1.1. Motivación

En la actualidad, en plena Era de la Información, las empresas de desarrollo de software tienen gran impacto en la sociedad, ya sea por la huella económica como por su influencia en la mejora de la calidad de vida.

Una parte importante del ciclo de vida de un producto de software es el mantenimiento, y es una tarea tediosa, pues se realiza sobre la aplicación finalizada, y un cambio puede implicara la revisión de gran parte del código fuente. Peor aún, dicha revisión puede recaer en un trabajador ajeno a la implementación original del código, ya sea porque el autor original ya no trabaja en la empresa o porque esté ocupado con otra tarea.

Muchas veces esta tarea no es difícil, sino larga, repetitiva y propensa a errores, que requiere unos recursos que podrían utilizarse en otra parte. Ésta es la motivación que subyace tras de este trabajo de fin de grado: demostrar que estas tareas pueden ser automatizadas.

1.2. Objetivos

Con la motivación descrita en el apartado anterior en mente, se presentan los siguientes objetivos:

- **Encontrar la herramienta más adecuada que permita desarrollar esta aplicación.** Hay que tener en cuenta los lenguajes de programación que tendrá que soportar el producto.
- **Desarrollar la aplicación.** Dado que se pretende modificar código fuente antes de su compilación, la aplicación deberá ser capaz de analizar el código y comprender su estructura. Como el abanico de posibilidades es muy amplio, se ha decidido desarrollar una herramienta de monitorización, que necesitará encontrar zonas específicas del código y luego modificarlas.
- **Añadir funcionalidades.** Una vez que se ha conseguido el objetivo anterior se deberá estudiar qué funcionalidades se le podrían añadir al producto de forma que cumpla mejor con los motivos de su desarrollo.

1.3. Estructura del documento

A continuación se muestra la estructura que va a seguir el documento, con una breve descripción del contenido de cada apartado.

1. **Introducción.** Contiene la motivación, los objetivos y la estructura del documento.
2. **Estado del arte.** Contiene las herramientas ya existentes relacionadas con la motivación de este TFG.
3. **Análisis y diseño.** Contiene la especificación de requisitos y las decisiones tomadas en el diseño del producto.
4. **Implementación e implantación.** Contiene los detalles de la implementación y la implantación de la herramienta.
5. **Evaluación.** Contiene el plan de pruebas con el que se ha verificado que la implementación de los requisitos ha sido correcta.
6. **Entorno socio-económico.** Contiene la planificación y el presupuesto del proyecto y su impacto socio económico.
7. **Conclusiones y trabajos futuros.** Contiene las conclusiones obtenidas del desarrollo del proyecto y una lista de posibles mejoras aplicables al producto.
8. **Anexo I: Repositorio.** Contiene la dirección web del repositorio donde pueden encontrarse todos los archivos generados durante el transcurso de este proyecto y las instrucciones de instalación del entorno de desarrollo.

2. Estado del arte

En este capítulo, se discutirá la situación actual de la transformación de código fuente a código fuente. Luego se presenta una comparación de los diferentes enfoques para el desarrollo de una herramienta que realice esa transformación.

2.1. Estudio de la actualidad

Actualmente sólo hay una herramienta que reescriba el código fuente con intenciones parecidas a las nuestras, y es un transpilador basado en ANTLR creado por un alumno de esta universidad [1]. Aunque el objetivo final de su proyecto era desarrollar una aplicación que actualizara automáticamente el código de las aplicaciones de móvil, es un ejemplo válido de las posibilidades que ofrece la reescritura automática de código fuente. Dado que su herramienta funciona con Java, también se pretende abarcar otros lenguajes para cubrir más mercado.

Apoyándonos en la Ilustración 1 podemos observar que los siguientes lenguajes más usados son C, Python y C++.

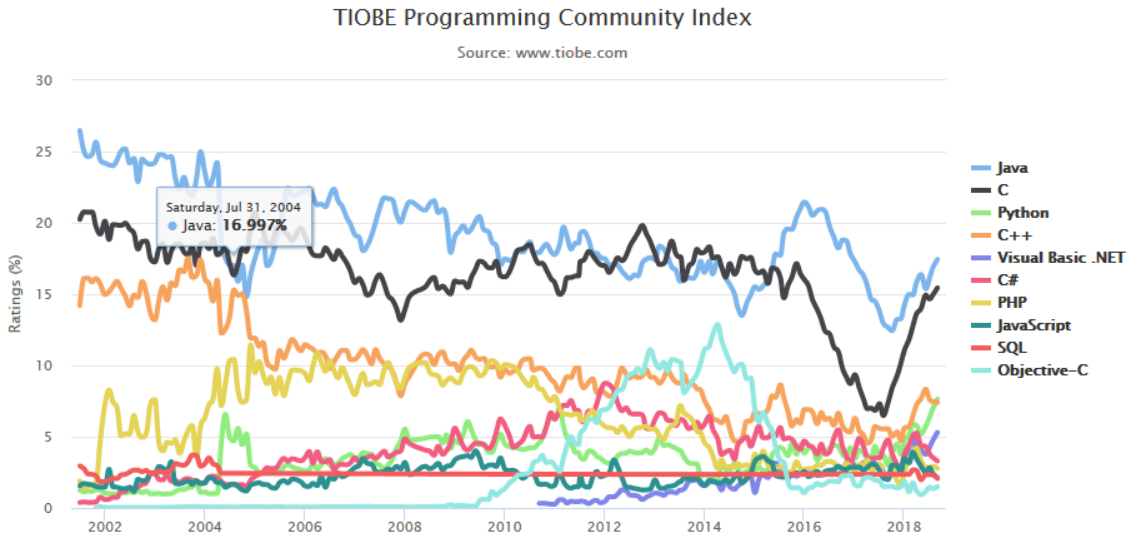

Ilustración 1 Gráfico de uso de los lenguajes [5]

Esto presenta la oportunidad de usar diferentes herramientas para diseñar nuestro transpilador.

Por otro lado, los IDE como IntelliJ[2] o Eclipse[3] ofrecen una función de refactorización, que es capaz de renombrar una variable o función y actualizar

las partes del código que hacen referencia a éstos, de forma que el código siga funcionando como antes del cambio.

Estos mismos IDE también, cuando se está desarrollando en Java, ofrecen la generación de los métodos get y set para los atributos privados de forma automática. Esto implica analizar el código fuente en busca de dichos atributos para poder crear una lista para que el usuario elija los que quiera generar.

Por último, pueden reescribir bucles con la intención de optimizarlos, o añadir o retirar imports automáticamente.

Dado que se ha elegido desarrollar una herramienta de monitorización hay que decir que es un mercado con muchos productos [4].

Lo que distingue a nuestra herramienta del resto es que el usuario no tiene que modificar su código para hacerlo compatible con la monitorización, nuestra herramienta ya se encarga de reescribirlo automáticamente.

Tampoco tendrá las limitaciones que tienen las herramientas que "observan" desde fuera, ya que al inyectarse en el código fuente, obtiene acceso a todas las variables necesarias.

2.2. Comparativa de las herramientas

Hay muchas opciones para generar parsers, como se puede ver aquí [6], pero no todas son apropiadas para la transformación de código fuente o compatibles con C y C++.

Para el desarrollo de nuestro producto estas son las tres mejores soluciones:

- Bison [7] y Flex [8]
- ANTLR [9]
- Clang [10]

A continuación, se proporciona una descripción breve de cada una y al final se proporciona una tabla comparativa.

2.2.1. Bison y Flex

Bison es un generador de analizadores sintácticos y Flex es un generador de analizadores léxicos. Ambas herramientas siguen el mismo flujo; empiezan con un fichero intermedio a partir del cual generan código en C. Sólo hay que compilar ambos ficheros .c para obtener la herramienta deseada.

La estructura del fichero de Flex es la siguiente:

```
definitions
%%
rules
%%
user code
```

Donde `definitions` es la sección donde se definen tokens con expresiones regulares; `rules` es la sección donde se especifican expresiones regulares y las acciones que realizará el analizador al encontrar una coincidencia; y `user code` es una sección de código en C que se copiará literalmente al fichero .c final.

La estructura del fichero de Bison es la siguiente:

```
%{
C declarations
%}
Bison declarations
%%
Grammar rules
%%
Additional C code
```

Donde `C declarations` es la sección donde se definen las macros y se declaran las funciones y variables que se van a usar en las acciones de la sección `Grammar rules`; `Bison declarations` es la sección donde se definen los símbolos terminales y no terminales, la precedencia entre símbolos con la misma prioridad; `Grammar rules` es la sección donde se especifican las reglas de la gramática y sus acciones; y `Additional C code` es una seccion igual que `user code` en Flex.

En cuanto a compatibilidad, Bison y Flex soportan C y C++. Sin embargo, las herramientas diseñadas con ellos sólo soportan un lenguaje a la vez y hacen uso de gramáticas LR.

2.2.2. ANTLR

ANTLR, ANother Tool for Languaje Recognition, es un generador de analizadores sintácticos con analizador léxico incorporado. Es capaz de generar código en Java a partir de un fichero con las gramáticas para el analizador léxico y el sintáctico.

La estructura del fichero es la siguiente:

```
grammar <filename>

/*
* Parser Rules
*/

Parser Rules

/*
* Lexer Rules
*/

Lexer Rules
```

Como se puede observar es bastante intuitivo, las reglas del analizador sintáctico van primero, con sus acciones correspondientes, y las del analizador léxico se definen después. Al principio del fichero hay que especificar un nombre para la gramática, que tiene que ser el mismo que el del fichero.

A diferencia de Bison y Flex, ANTLR usa gramáticas LL, que aunque son menos poderosas que las LR son más fáciles de escribir, y soporta muchos más lenguajes. Gracias a este repositorio [11] no es necesario escribir la gramática entera del lenguaje que se vaya a usar. Aun así, hay que crear un analizador sintáctico para cada lenguaje.

2.2.3. Clang

Clang es el "front end" para los lenguajes en la familia de C (C, C++, Objective C/C++, OpenCL, CUDA, and RenderScript) del compilador LLVM [12].

También provee de la infraestructura necesaria para desarrollar herramientas que trabajen con código fuente en esos lenguajes, mediante tres interfaces [13].

A diferencia de los anteriores, para desarrollar una herramienta no es necesario especificar ninguna gramática, ya que se trabaja sobre el AST que genera Clang, por lo que el trabajo es a un nivel más abstracto. Dependiendo de la interfaz elegida para comunicarse con Clang, el usuario puede interactuar con el AST con mayor o menor profundidad o hasta desarrollar heramientas capaces de funcionar en cualquier máquina, incluso sin que Clang ni LLVM estén instalados.

Además, también ofrece libertad al usuario para escoger en qué lenguaje quiere desarrollar la herramienta, y esta funcionará en todos los lenguajes especificados al principio de este apartado.

2.2.4. Tabla comparativa

A continuación se muestra la tabla con la comparación entre las herramientas en los campos más relevantes para este proyecto:

	Bison y Flex	ANTLR	Clang
Soporta C	SI	SI	SI
Soporta C++	SI	SI	SI
Numero de ficheros	2	1	1
Nivel de abstracción	BAJO	BAJO	ALTO
Herramienta multi-lenguaje	NO	NO	SI
Lenguajes de desarrollo	C/C++	JAVA	TODOS*

Tabla 1 Comparativa de herramientas

* Para obtener todas las funcionalidades es necesario desarrollar la herramienta en C++.

Dado que parece que Clang es una herramienta más versátil se ha decidido emplearla para desarrollar nuestro producto.

3. Análisis y diseño

Ahora se procede a exponer los resultados del análisis de requisitos, y los casos de uso y demás decisiones de diseño tomadas.

3.1. Análisis

En este apartado se muestra la especificación de requisitos, siguiendo el siguiente formato:

ID	RX-NN		
Título			
Descripción			
Claridad		Necesidad	
Verificabilidad		Estabilidad	

Tabla 2 Ejemplo de especificación de requisitos

Donde:

- **RX-NN** es la etiqueta de identificación única del requisito, empleándose **RF** o **RNF** en lugar de **RX** si se trata de un requisito funcional o uno no funcional, respectivamente, y el número de requisito en lugar de **NN**.
- **Título** es el nombre del requisito.
- **Descripción** es el cuerpo que define el requisito, siendo lo más breve y claro posible.
- **Claridad** es una medida de cómo de ambiguo es el requisito. Puede ser **Alta** (poco ambiguo), **Media** (parcialmente ambiguo) o **Baja** (muy ambiguo).
- **Necesidad** es una medida de cómo de necesario es que el producto cumpla con este requisito. Puede ser **Alta** (esencial), **Media** (preferible) o **Baja** (opcional).
- **Verificabilidad** es una medida de cómo de fácil es verificar que se cumple el requisito. Puede ser **Alta** (fácil), **Media** (normal) o **Baja** (difícil).
- **Estabilidad** es la medida de cuánto tiempo permanecerá el requisito en el ciclo de vida del producto. Puede ser **Alta** (mucho tiempo), **Media** (algo de tiempo) o **Baja** (poco tiempo)

3.1.1. Requisitos funcionales

ID	RF-01		
Título	Configuración de monitorización		
Descripción	Las funcionalidades de monitorización de la aplicación deben ser configurables		
Claridad	Alta	**Necesidad**	Alta
Verificabilidad	Alta	**Estabilidad**	Alta

Tabla 3 RF-01

ID	RF-02		
Título	Monitorización de tiempo		
Descripción	La aplicación debe inyectar el código necesario para monitorizar los tiempos de ejecución de los métodos del código fuente original.		
Claridad	Alta	**Necesidad**	Alta
Verificabilidad	Alta	**Estabilidad**	Alta

Tabla 4 RF-02

ID	RF-03		
Título	Monitorización de I/O		
Descripción	La aplicación debe inyectar el código necesario para monitorizar los accesos de lectura/escritura de ficheros de los métodos del código fuente original.		
Claridad	Alta	**Necesidad**	Alta
Verificabilidad	Alta	**Estabilidad**	Alta

Tabla 5 RF-03

ID	RF-04		
Título	Monitorización de memoria		
Descripción	La aplicación debe inyectar el código necesario para monitorizar las reservas/liberaciones de memoria de los métodos del código fuente original.		
Claridad	Alta	**Necesidad**	Alta
Verificabilidad	Alta	**Estabilidad**	Alta

Tabla 6 RF-04

ID	RF-05		
Título	Compatibilidad con C		
Descripción	La aplicación debe poder trabajar con código fuente en C.		
Claridad	Alta	**Necesidad**	Alta
Verificabilidad	Alta	**Estabilidad**	Alta

Tabla 7 RF-05

ID	RF-06		
Título	Compatibilidad con C++		
Descripción	La aplicación debe poder trabajar con código fuente en C++.		
Claridad	Alta	**Necesidad**	Alta
Verificabilidad	Alta	**Estabilidad**	Alta

Tabla 8 RF-06

ID	RF-07		
Título	Fichero de salida		
Descripción	La aplicación debe generar un fichero con los resultados de la monitorización.		
Claridad	Alta	**Necesidad**	Alta
Verificabilidad	Alta	**Estabilidad**	Media

Tabla 9 RF-07

ID	RF-08		
Título	Configuración del fichero de salida		
Descripción	El nombre y la ubicación del fichero de salida deben ser configurables		
Claridad	Alta	**Necesidad**	Alta
Verificabilidad	Alta	**Estabilidad**	Alta

Tabla 10 RF-08

3.1.2. Requisitos no funcionales

ID	RNF-01		
Título	Modo de ejecución		
Descripción	La aplicación debe ser ejecutada desde el terminal		
Claridad	Alta	**Necesidad**	Alta
Verificabilidad	Alta	**Estabilidad**	Alta

Tabla 11 RNF-01

ID	RNF-02		
Título	Modo de configuración		
Descripción	Las opciones de configuración de la aplicación deben ser especificadas como argumentos del comando de ejecución		
Claridad	Alta	**Necesidad**	Alta
Verificabilidad	Alta	**Estabilidad**	Media

Tabla 12 RNF-02

ID	RNF-03		
Título	Lenguaje de desarrollo		
Descripción	La aplicación debe ser desarrollada en lenguaje C++		
Claridad	Alta	**Necesidad**	Alta
Verificabilidad	Alta	**Estabilidad**	Alta

Tabla 13 RNF-03

ID	RNF-04		
Título	Compatibilidad		
Descripción	La aplicación debe ser compatible con las distribuciones compatibles con la familia Debian		
Claridad	Alta	**Necesidad**	Alta
Verificabilidad	Alta	**Estabilidad**	Alta

Tabla 14 RNF-04

3.1.3. Matriz de relación

Esta es la matriz que relaciona los requisitos entre ellos:

ID	RF-01	RF-02	RF-03	RF-04	RF-05	RF-06	RF-07	RF-08	RNF-01	RNF-02	RNF-03	RNF-04
RF-01		X	X	X			X			X		
RF-02	X						X			X		
RF-03	X						X			X		
RF-04	X						X			X		
RF-05												
RF-06												
RF-07	X	X	X	X				X		X		
RF-08							X			X		
RNF-01												
RNF-02	X	X	X	X			X	X				
RNF-03												
RNF-04												

Tabla 15 Matriz de relación de requisitos

3.1.4. Marco regulador

Finalmente hay dos leyes que afectan a la aplicación, la ley de protección de datos [14] y la ley de protección intelectual [15].

La ley de protección de datos no afecta a nuestra herramienta porque no recopila ningún dato, y por tanto, no necesita informar ni solicitar permiso para tratar los datos.

En cuanto a la ley de protección intelectual, Clang [16] utiliza la licencia BSD [17], que permite usar el código tanto para software libre como privativo.

3.2. Diseño

Una vez expuestas las especificaciones de la herramienta, se procede con las decisiones que se han tomado para el diseño del producto.

3.2.1. Casos de uso

El primer caso se produce cuando el usuario ejecuta la herramienta sin especificar ninguna opción ni proporcionar argumentos, por lo que se le mostrarán las instrucciones para la correcta utilización del producto.

Ilustración 2 Ejecución sin argumentos

El segundo caso ocurre cuando el usuario ejecuta la herramienta con la opción de ayuda, obteniendo un menú con las diversas opciones disponibles.

Ilustración 3 Ejecución del comando de ayuda

El tercer caso se produce al ejecutar la herramienta con argumentos, pero sin especificar las opciones de monitorización deseadas, que aunque es una manera correcta de invocarla, no producirá ningún cambio en el código fuente.

Ilustración 4 Ejecución sin opciones

Por último al ejecutar la herramienta con una o más opciones de monitorización y proporcionándole el código fuente a modificar por los argumentos, se pueden dar dos situaciones. En caso de éxito se generarán ficheros con el código fuente modificado, y en caso de error se mostrará por pantalla la razón del mismo.

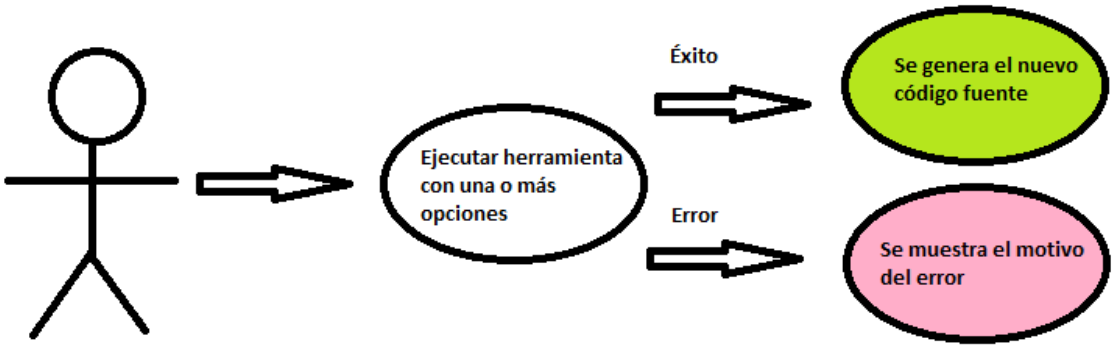

Ilustración 5 Ejecución con una o más opciones

3.2.2. Decisiones adicionales

La herramienta Clang ofrece tres interfaces para el desarrollo de aplicaciones:

- LibClang: LibClang es una interfaz compatible hacia atrás, por lo que una herramienta creada con esta interfaz no necesitaría modificarse al actualizarse Clang. También permite que la herramienta pueda ser implementada en un lenguaje diferente a C++, aunque no da control total sobre el AST que genera Clang.
- Clang Plugins: Los plugins son librerías dinámicas que se cargan en tiempo de ejecución por el compilador. Ofrecen acceso a todas las funcionalidades de Clang, pero las herramientas hechas con esta interfaz no pueden usarse fuera del entorno de desarrollo de Clang.
- LibTooling: Es una interfaz en C++ que tiene acceso a todas las funcionalidades de Clang y puede compilarse de forma que pueda usarse en cualquier máquina, incluso si esta no tiene el entorno de desarrollo de Clang. Permite ejecutar la herramienta sobre uno o más ficheros de forma manual, a diferencia de los plugins, que pueden ser automatizados para actuar cuando hay un cambio de dependencias.

La interfaz elegida es LibTooling, ya que no obliga al usuario final a tener instalado en entorno de desarrollo de Clang, y tiene todas las funcionalidades.

También se ha decidido que la herramienta sólo introduzca llamadas a funciones que se encontrarán en una librería externa que se proporcionará con la herramienta.

4. Implementación e implantación

A continuación se detallan los aspectos más notables del desarrollo de la aplicación, seguidos por la especificaciones de la máquina empleada y los requisitos mínimos para instalar el entorno de desarrollo.

4.1. Implementación

En este apartado se van a detallar los obstáculos encontrados durante la implementación de la herramienta.

Lo primero es destacar que pese a existir una documentación oficial y reciente [18], parece no haber sido revisada en detalle, ya que algunos pasos no son realizables, como la instalación de Cmake que utiliza un repositorio que ya no da servicio. También cabe destacar que otros pasos no funcionan por la falta de pasos intermedios no especificados.

Además, pese a la razonable cantidad de RAM de que disponía la máquina donde se estaba instalando el entorno de desarrollo, la instalación se detenía frecuentemente por falta de memoria, llegando a ser necesaria la adquisición de una máquina nueva con más memoria. Aun así la instalación falló muchas veces, pero por suerte al reintentar reanudaba en el punto donde la instalación anterior paró.

Tras salvar todos estos obstáculos se procedió a compilar una herramienta que simplemente mostrara el AST que generaba Clang, y se descubrió que este proceso requiere de varios minutos, ya que se está compilando todo el código adicional necesario para que la herramienta funcione sola. Aunque comprensible, esto retrasó mucho la implementación pues, aunque se trató de acumular los cambios para reducir el número de compilaciones, hubo muchas compilaciones.

La implementación se dividió en varios prototipos, cada uno de mayor complejidad que el anterior, mientras se estudiaban las diversas soluciones a los problemas propuestos.

El primer prototipo tenía como objetivo comprobar la correcta instalación del entorno de desarrollo y la capacidad de Clang de generar un AST:

Para este código:

```
1 void do_math(int *x) {
2   if(*x < 5)
3     *x += 5;
4   else
5     *x += 10;
6   if(*x > 20){
7     *x -= 11;
8   }
9 }
10
11 int main(void) {
12     int result = -1, val = 4;
13     do_math(&val);
14     return result;
15 }
```

Ilustración 6 Código antes de analizar

Éste es el resultado:

```
alejandro@alejandro-LAPTOP ~/clang-llvm/build $ bin/injector test-files/injectorTestV0.c --
** Creating AST consumer for: /home/alejandro/clang-llvm/build/test-files/injectorTestV0.c
FunctionDecl 0x4aa5d20 </home/alejandro/clang-llvm/build/test-files/injectorTestV0.c:1:1, line:9:1> line:1:6 do_math 'void (int *)'
|-ParmVarDecl 0x4aa5c60 <col:14, col:19> col:19 used x 'int *'
`-CompoundStmt 0x4aa6238 <col:22, line:9:1>
  |-IfStmt 0x4aa6038 <line:2:3, line:5:11>
  | |-<<<NULL>>>
  | |-<<<NULL>>>
  | |-BinaryOperator 0x4aa5ea0 <line:2:6, col:11> 'int' '<'
  | | |-ImplicitCastExpr 0x4aa5e88 <col:6, col:7> 'int' <LValueToRValue>
  | | | `-UnaryOperator 0x4aa5e48 <col:6, col:7> 'int' lvalue prefix '*' cannot overflow
  | | |   `-ImplicitCastExpr 0x4aa5e30 <col:7> 'int *' <LValueToRValue>
  | | |     `-DeclRefExpr 0x4aa5e08 <col:7> 'int *' lvalue ParmVar 0x4aa5c60 'x' 'int *'
  | | `-IntegerLiteral 0x4aa5e68 <col:11> 'int' 5
  | |-CompoundAssignOperator 0x4aa5f48 <line:3:5, col:11> 'int' '+=' ComputeLHSTy='int' ComputeResultTy='int'
  | | |-UnaryOperator 0x4aa5f08 <col:5, col:6> 'int' lvalue prefix '*' cannot overflow
  | | | `-ImplicitCastExpr 0x4aa5ef0 <col:6> 'int *' <LValueToRValue>
  | | |   `-DeclRefExpr 0x4aa5ec8 <col:6> 'int *' lvalue ParmVar 0x4aa5c60 'x' 'int *'
  | | `-IntegerLiteral 0x4aa5f28 <col:11> 'int' 5
  | |-CompoundAssignOperator 0x4aa6000 <line:5:5, col:11> 'int' '+=' ComputeLHSTy='int' ComputeResultTy='int'
  | | |-UnaryOperator 0x4aa5fc0 <col:5, col:6> 'int' lvalue prefix '*' cannot overflow
  | | | `-ImplicitCastExpr 0x4aa5fa8 <col:6> 'int *' <LValueToRValue>
  | | |   `-DeclRefExpr 0x4aa5f80 <col:6> 'int *' lvalue ParmVar 0x4aa5c60 'x' 'int *'
  | | `-IntegerLiteral 0x4aa5fe0 <col:11> 'int' 10
  |-IfStmt 0x4aa6200 <line:6:3, line:8:3>
  | |-<<<NULL>>>
  | |-<<<NULL>>>
  | |-BinaryOperator 0x4aa6108 <line:6:6, col:11> 'int' '>'
  | | |-ImplicitCastExpr 0x4aa60f0 <col:6, col:7> 'int' <LValueToRValue>
  | | | `-UnaryOperator 0x4aa60b0 <col:6, col:7> 'int' lvalue prefix '*' cannot overflow
  | | |   `-ImplicitCastExpr 0x4aa6098 <col:7> 'int *' <LValueToRValue>
  | | |     `-DeclRefExpr 0x4aa6070 <col:7> 'int *' lvalue ParmVar 0x4aa5c60 'x' 'int *'
  | | `-IntegerLiteral 0x4aa60d0 <col:11> 'int' 20
  | `-CompoundStmt 0x4aa61e8 <col:14, line:8:3>
  |   `-CompoundAssignOperator 0x4aa61b0 <line:7:5, col:11> 'int' '-=' ComputeLHSTy='int' ComputeResultTy='int'
  |     |-UnaryOperator 0x4aa6170 <col:5, col:6> 'int' lvalue prefix '*' cannot overflow
  |     | `-ImplicitCastExpr 0x4aa6158 <col:6> 'int *' <LValueToRValue>
  |     |   `-DeclRefExpr 0x4aa6130 <col:6> 'int *' lvalue ParmVar 0x4aa5c60 'x' 'int *'
  |     `-IntegerLiteral 0x4aa6190 <col:11> 'int' 11
  `-<<<NULL>>>
FunctionDecl 0x4aa6328 </home/alejandro/clang-llvm/build/test-files/injectorTestV0.c:11:1, line:15:1> line:11:5 main 'int (void)'
`-CompoundStmt 0x4aa66b8 <col:16, line:15:1>
  |-DeclStmt 0x4aa6560 <line:12:5, col:29>
  | |-VarDecl 0x4aa6410 <col:5, col:19> col:9 used result 'int' cinit
  | | `-UnaryOperator 0x4aa6490 <col:18, col:19> 'int' prefix '-'
  | |   `-IntegerLiteral 0x4aa6470 <col:19> 'int' 1
  | `-VarDecl 0x4aa64c8 <col:5, col:28> col:22 used val 'int' cinit
  |   `-IntegerLiteral 0x4aa6528 <col:28> 'int' 4
  |-CallExpr 0x4aa6630 <line:13:5, col:17> 'void'
  | |-ImplicitCastExpr 0x4aa6618 <col:5> 'void (*)(int *)' <FunctionToPointerDecay>
  | | `-DeclRefExpr 0x4aa6578 <col:5> 'void (int *)' Function 0x4aa5d20 'do_math' 'void (int *)'
  | `-UnaryOperator 0x4aa65c8 <col:13, col:14> 'int *' prefix '&' cannot overflow
  |   `-DeclRefExpr 0x4aa65a0 <col:14> 'int' lvalue Var 0x4aa64c8 'val' 'int'
  |-ReturnStmt 0x4aa66a0 <line:14:5, col:12>
  | `-ImplicitCastExpr 0x4aa6688 <col:12> 'int' <LValueToRValue>
  |   `-DeclRefExpr 0x4aa6660 <col:12> 'int' lvalue Var 0x4aa6410 'result' 'int'
alejandro@alejandro-LAPTOP ~/clang-llvm/build $
```

Ilustración 7 El AST resultante

Cabe destacar que los tutoriales oficiales no profundizan mucho más allá de este punto. Si bien es cierto que existe documentación de todas las clases que componen Clang, normalmente no detalla nada más que las relaciones de una clase con las otras y los métodos de los que dispone, por lo que se ha tenido que hacer un uso extensivo de motores de búsqueda y de Stack Overflow.

El objetivo del segundo prototipo era detectar el inicio y fin de todas las declaraciones de funciones, y el de las sentencias if. Para el código anterior, éste es el resultado:

```
1 // Begin function do_math returning void
2 void do_math(int *x) {
3   if(*x < 5)
4     // the 'if' part
5     *x += 5;
6     // end of the 'if' part
7
8   else
9     // the 'else' part
10    *x += 10;
11    // end of the 'else' part
12
13  if(*x > 20)// the 'if' part
14  {
15    *x -= 11;
16  }
17
18 // end of the 'if' part
19 }
20 // End function do_math
21
22 // Begin function main returning int
23 int main(void) {
24    int result = -1, val = 4;
25    do_math(&val);
26    return result;
27 }
28 // End function main
```

Ilustración 8 El código tras la modificación

En este prototipo nos encontramos con un obstáculo: Clang ofrece dos funciones, getLocStart() y getLocEnd() para determinar el inicio y fin de la zona en la que se está trabajando. La primera funciona perfectamente, pero la segunda apunta siempre un token por delante de donde se esperaría. Si se usara getLocEnd() en el if, la sección *//end of the 'if' part* se escribiría entre el '+=' y el '5;'.

Parece ser un bug que Clang arrastra desde hace un tiempo, pero que se ha podido solucionar con un poco de astucia.

```
TheRewriter.InsertText(Then->getLocStart(), "// the 'if' part\n", true, true);
SourceLocation end = Then->getLocEnd();
int offset = Lexer::MeasureTokenLength(end,
                                       TheRewriter.getSourceMgr(),
                                       TheRewriter.getLangOpts()) + 1;
SourceLocation realEnd = end.getLocWithOffset(offset);
TheRewriter.InsertText(realEnd, "\n// end of the 'if' part\n", true, true);
```

Ilustración 9 El arreglo de getLocEnd()

El tercer prototipo estaba diseñado para encontrar llamadas a una función específica - en este caso la función malloc - y especificar qué función lo llamó. Aquí nos encontramos el siguiente obstáculo: encontrar los nodos del árbol correspondientes a llamadas a malloc es fácil, pero Clang no nos da ninguna forma de obtener al llamador desde ellos. La solución al final radica en el orden en el que Clang recorre los nodos: tanto el llamador como el llamado son nodos de tipo FunctionDecl, pero el llamador siempre se recorre primero y, a diferencia del llamado, representa una función con cuerpo por lo que una simple llamada a la función hasBody() y el uso de una variable global nos permite obtener ambos.

```
alejandro@alejandro-LAPTOP ~/clang-llvm/build $ bin/injector test-files/injectorTestV3.c --
** Creating AST consumer for: /home/alejandro/clang-llvm/build/test-files/injectorTestV3.c
** Found call for function: malloc in function; do_math
** Found call for function: printf in function; do_math
** Found call for function: do_math in function; main
** EndSourceFileAction for: /home/alejandro/clang-llvm/build/test-files/injectorTestV3.c
```

Ilustración 10 Salida del prototipo por el terminal

```c
#include <stdio.h>
#include <stdlib.h>
// Begin function do_math returning void
void do_math(int *x) {
  if(*x < 5)
    // the 'if' part
    *x += 5;
    // end of the 'if' part

  else
    // the 'else' part
    *x += 10;
    // end of the 'else' part

  if(*x > 20)// the 'if' part
  {
    *x -= 11;
  }

    // end of the 'if' part
        char *str = (char *) malloc(11);// PLACEHOLDER

    str = "Hello World";
    printf("%s\n",str);
}
// End function do_math

// Begin function main returning int
int main(void) {
    int result = -1, val = 4;
    do_math(&val);
    return result;
}
// End function main
```

Ilustración 11 El código modificado tras el tercer prototipo

Con los conocimientos obtenidos del desarrollo de los tres prototipos se procedió a implementar la herramienta sin encontrar más obstáculos.

4.2. Implantación

Dado que la herramienta ha sido compilada de forma que todo el código que necesite sea incluido en ella, el único requerimiento para utilizar esta herramienta es utilizar una máquina con un sistema operativo compatible con la familia Debian.

De todas formas, por si hubiera algún problema, el entorno de desarrollo ha sido Clang 8 en un sistema operativo Linux Mint 18.1 en un portátil HP Notebook 250 G6 Intel Core i5-7200U con 8GB de RAM, 256GB de disco duro de estado sólido y pantalla de 15.6".

Los requerimientos mínimos para instalar el entorno de desarrollo son los siguientes:

- **60 GB de espacio en el disco duro.** El entorno de desarrollo requiere la instalación de las herramientas LLVM y Clang. The development environment requires the download and build of the LLVM and Clang tools. The downloaded files occupy less than 5GB, but the build files folder increases its weigh rapidly during the installation process.
- **8 GB of RAM memory.** Using less than this amount will increase installation times and failures way over reason.

5. Evaluación

Para la correcta evaluación de la herramienta se han creado las pruebas de forma que todos los requisitos queden verificados.

5.1. Plan de pruebas

ID	Prueba-XX
Descripción	
Precondiciones	
Procedimiento	
Postcondiciones	
Verificado	
Requisitos	

Tabla 16 Ejemplo de prueba

Donde:

- **ID** es el identificador de la prueba. Sigue el formato Prueba-XX donde XX es un número entre 0 y 99.
- **Descripción** es un resumen de los objetivos de este test.
- **Precondiciones** son las condiciones que han de cumplirse antes de ejecutar esta prueba.
- **Procedimiento** es una aclaración del proceso a seguir para realizar la prueba.
- **Postcondiciones** son las condiciones que han de cumplirse al finalizar el test.
- **Verificado** es **SI** o **NO**, dependiendo de si el resultado de la prueba ha sido satisfactorio o no, respectivamente.
- **Requisitos** es la lista de requisitos verificados.

ID	Prueba-01
Descripción	Monitorización de tiempo
Precondiciones	El terminal debe encontrarse en el directorio en el que se encuentra la herramienta.
Procedimiento	Ejecutar la herramienta especificando la opción -time: ./injector -time <nombredelcodigofuente> --
Postcondiciones	El código fuente modificado contiene el código necesario para monitorizar el tiempo de ejecución de los métodos.
Verificado	**SI**
Requisitos	RF-01, RF-02, RF-07, RF-08, RNF-01, RNF-02, RNF-03, RNF-04

Tabla 17 Prueba-01

ID	Prueba-02
Descripción	Monitorización de I/O
Precondiciones	El terminal debe encontrarse en el directorio en el que se encuentra la herramienta.
Procedimiento	Ejecutar la herramienta especificando la opción -io: ./injector -io <nombredelcodigofuente> --
Postcondiciones	El código fuente modificado contiene el código necesario para monitorizar las operaciones de entrada/salida.
Verificado	**SI**
Requisitos	RF-01, RF-03, RF-07, RF-08, RNF-01, RNF-02, RNF-03, RNF-04

Tabla 18 Prueba-02

ID	Prueba-03
Descripción	Monitorización de memoria
Precondiciones	El terminal debe encontrarse en el directorio en el que se encuentra la herramienta.
Procedimiento	Ejecutar la herramienta especificando la opción -mem: ./injector -mem <nombredelcodigofuente> --
Postcondiciones	El código fuente modificado contiene el código necesario para monitorizar las operaciones de reserva/liberación de memoria.
Verificado	**SI**
Requisitos	RF-01, RF-04, RF-07, RF-08, RNF-01, RNF-02, RNF-03, RNF-04

Tabla 19 Prueba-03

ID	Prueba-04
Descripción	Compatibilidad con C
Precondiciones	El terminal debe encontrarse en el directorio en el que se encuentra la herramienta.
Procedimiento	Ejecutar la herramienta especificando cualquier opción de monitorización sobre un fichero en C: ./injector -io <nombredelcodigofuente.c> --
Postcondiciones	El código fuente modificado contiene el código necesario para monitorizar las operaciones de entrada/salida.
Verificado	**SI**
Requisitos	RF-05, RF-08, RNF-01, RNF-02, RNF-03, RNF-04

Tabla 20 Prueba-04

ID	Prueba-05
Descripción	Compatibilidad con C++
Precondiciones	El terminal debe encontrarse en el directorio en el que se encuentra la herramienta.
Procedimiento	Ejecutar la herramienta especificando cualquier opción de monitorización sobre un fichero en C++: ./injector -mem <nombredelcodigofuente.cpp> --
Postcondiciones	El código fuente modificado contiene el código necesario para monitorizar las operaciones de reserva/liberación de memoria.
Verificado	**SI**
Requisitos	RF-06, RF-08, RNF-01, RNF-02, RNF-03, RNF-04

Tabla 21 Prueba-05

5.2. Matriz de trazabilidad

Esta es la matriz de trazabilidad:

ID	RF-01	RF-02	RF-03	RF-04	RF-05	RF-06	RF-07	RF-08	RNF-01	RNF-02	RNF-03	RNF-04
Prueba-01	X	X					X	X	X	X	X	X
Prueba-02	X		X				X	X	X	X	X	X
Prueba-03	X			X			X	X	X	X	X	X
Prueba-04					X		X	X	X	X	X	X
Prueba-05						X	X	X	X	X	X	X

Tabla 22 Matriz de trazabilidad

6. Entorno socio-económico

Ahora se procede con la especificación de la planificación y el presupuesto del proyecto. Después se analizará el impacto socio-económico que tendría el desarrollo de este TFG.

6.1. Planificación

A continuación se detallará la planificación prevista para el desarrollo de las diferentes tareas que han sido necesarias para el desarrollo de la herramienta. Para ello se va a hacer uso de dos diagramas de Gantt: uno con las fechas de finalización estimadas y otro con las duraciones reales.

Como se puede observar por la diferencia entre las ilustraciones 12 y 13 se han producido varios retrasos que han afectado a la fecha de finalización estimada del proyecto. Cabe destacar que el retraso más notable fue en la preparación del entorno de desarrollo, ya que no especificaba por ninguna parte los requisitos del sistema mínimos para la instalación. Por esto, fue necesario obtener una máquina nueva en la que instalarlo y, pese a la mejora en cuanto a cantidad de RAM y el uso de un SSD, hicieron falta varios días más de los previstos para empezar a desarrollar la herramienta.

También se produjeron retrasos durante la implementación por la escasez de documentación, lo que hacía muy difícil resolver los obstáculos que se encontraron al ir incrementando las funcionalidades del prototipo.

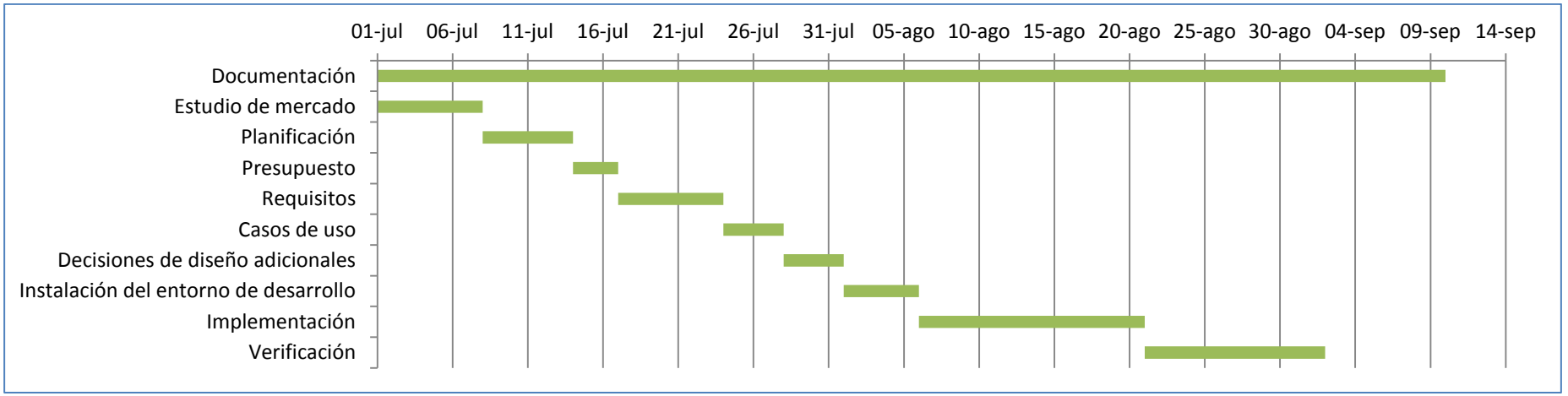

Ilustración 12 Planificación estimada

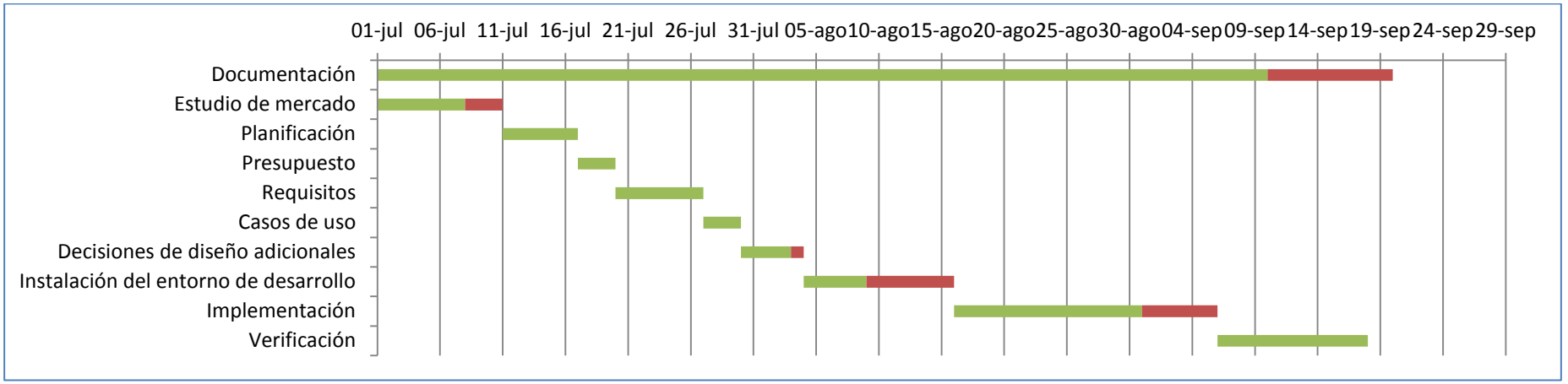

Ilustración 13 Planificación real

6.2. Presupuesto

Para el cálculo del presupuesto primero hay que obtener las horas del proyecto en la estimación dada en la planificación:

Planificación estimada	
Horas	312
Semanas	10.2857143
Horas/Semana	30.3333333
Horas/Mes	133.714286

Tabla 23 Planificación estimada

Después se calcula el coste a la empresa de cada empleado. Con la información obtenida de para los salarios y para la cotización a la Seguridad Social y suponiendo contratos indefinidos, 160 horas al mes, trabajo de bajo riesgo y ninguna hora extra:

Sueldo bruto		
Rol	Sueldo anual	Sueldo mensual
	€	€
Jefe de proyecto	35,769	2,980.75
Analista	28,853	2,404.42
Programador	26,396	2,199.67
QA	26,396	2,199.67

Tabla 24 Sueldo bruto por rol

Cotización a la Seguridad Social							
Rol	Contingencias comunes y profesionales		Desempleo		FOGASA y Formación Profesional		Total
	%	€	%	€	%	€	€
Jefe de proyecto	23.6+1	733.26	5.5	163.94	0.2+0.6	23.85	921.05
Analista	23.6+1	591.49	5.5	132.24	0.2+0.6	19.24	742.97
Programador	23.6+1	541.12	5.5	120.98	0.2+0.6	17.60	679.70
QA	23.6+1	541.12	5.5	120.98	0.2+0.6	17.60	679.70

Tabla 25 Cotización a la Seguridad Social por rol

Con estos datos se puede calcular el coste por hora de cada empleado a la empresa:

Coste a la empresa		
Rol	Coste mensual	Coste por hora
	€	€
Jefe de proyecto	3,901.80	24.39
Analista	3,147.39	19.67
Programador	2,879.37	18
QA	2,879.37	18

Tabla 26 Coste a la empresa por rol

Ahora se le asigna a cada rol las horas estimadas del proyecto:

Coste del personal			
Rol	Coste por hora	Horas asignadas	Total
	€	horas	€
Jefe de proyecto	24.39	72	1,756.08
Analista	19.67	100	1,967
Programador	18	85	1,530
QA	18	55	990
Coste personal			6,243.08

Tabla 27 Coste total por horas asignadas

Una vez obtenidos los costes de personal, se procede con el coste de los equipos necesarios para el desarrollo del producto:

Coste del equipo				
Equipo	Coste	Amortización	Uso	Total
	€	años	semanas	€
HP Notebook 250 G6 Intel Core i5-7200U/8GB/256GB SSD/15.6"	558.41	2	10.29	55.25
Teclado Logitech K120	11.99	2	10.29	1.19
Ratón Logitech M90	9.90	2	10.29	0.98
Monitor Asus VS229H-P 21.5"	109	2	10.29	10.78
Coste del equipo				68.2

Tabla 28 Coste del equipo

Ahora, con el coste de personal y el de equipo podemos obtener el coste base al que aplicarle el margen de riesgo, el de beneficio y el IVA para conseguir el coste del proyecto:

Coste del proyecto								
Base	Margen de riesgo		Margen de beneficio		Coste sin IVA	IVA		Total
€	%	€	%	€	€	%	€	€
6,311.28	10	631.13	15	946.69	7,889.1	21	1,656.71	**9545.81**

Tabla 29 Coste total del proyecto

6.3. Impacto socio-económico

Como ya se mencionó en el apartado de motivación, las empresas se ven obligadas a gastar una cantidad importante de recursos en el mantenimiento de sus productos de software.

Este TFG es una demostración de la capacidad para desarrollar herramientas que realicen dicho mantenimiento de forma automática en la actualidad.

Cada tarea que se automatice liberará recursos que podrán dedicarse a la realización de otros proyectos lucrativos para la empresa.

Otro resultado de esta automatización es que, al tener más recursos, las empresas aprovechen para mejorar el servicio que proporcionan sus productos, lo cual repercutirá positivamente en los clientes y usuarios.

Por ello se observa que el desarrollo de aplicaciones con el objetivo de automatizar el mantenimiento de los productos de software **tendrá un impacto positivo,** tanto económicamente para las empresas, como en la calidad de vida.

7. Conclusiones y trabajos futuros

Por último se muestran las conclusiones obtenidas del desarrollo de este proyecto, tanto del producto como personales, y se proponen varias ideas que podrían ser incorporadas al mismo.

7.1. Conclusiones

En cuanto al producto cabe destacar la cantidad de posibilidades que ofrece la modificación automática de código, y su poder. Un ejemplo muy común en la actualidad, tanto que cuesta pensar que en algún momento esta funcionalidad no existiera, es la refactorización.

Con Clang todas estas posibilidades están al alcance. Dado que genera un árbol sintáctico abstracto, se puede implementar los cambios que se deseen con muy poco código. Y, al tratar los datos desde un nivel más abstracto que otras herramientas, resulta transparente para el usuario.

Este proyecto ha demostrado la facilidad de desarrollar aplicaciones en este campo usando Clang: con muy pocas líneas de código se pueden obtener herramientas muy poderosas. Por ejemplo el primer prototipo implementado tiene 58 líneas, el segundo tiene 156 y el tercero 180. Cabe destacar el tercer prototipo tiene todas las funcionalidades del prototipo anterior y que la herramienta final no tiene muchas líneas más.

De hecho una vez aprendido el comportamiento general de las aplicaciones que usan la interfaz de LibTooling no requiere mucho tiempo crearlas desde cero o reformar código ya existente.

De forma personal este TFG ha resultado muy gratificante, ya que el autor siempre ha encontrado fascinante el trabajo de los compiladores, y su capacidad para entender e interpretar el código fuente. Dado que la herramienta propuesta en este TFG no cambia el nivel de abstracción del lenguaje, en realidad es un transpilador, pero todos los transpiladores son compiladores al fin y al cabo.

Por último, el desarrollo de este proyecto ha ayudado a reforzar los conocimientos del autor sobre todo en C++, ya que no es un lenguaje muy solicitado durante el desarrollo de la carrera.

7.2. Trabajos Futuros

A continuación se presentan algunas ideas que podrían incorporar a la herramienta desarrollada en este proyecto:

- **Selección de los funciones a monitorizar.** Actualmente si se especifica la funcionalidad de monitorización, todos las funciones encontradas en el código fuente son monitorizadas. Esto no siempre es deseable y permitir dicha selección sería una mejora.
- **Refactorizar.** Aunque esto puede hacerse fácilmente en un editor moderno, añadir esta funcionalidad también sería beneficioso.
- **Selección del nombre del fichero de salida.** En el estado actual del producto, todos los ficheros generados tienen el mismo nombre que su original con la adición de una "M" al final. Por ello estaría bien permitir al usuario decidir el nombre del fichero de salida.
- **Utilización de un fichero de configuración.** Conforme se van añadiendo funcionalidades con sus correspondientes opciones, se va incrementando la complejidad del comando de ejecución, haciéndolo propenso a errores. Por eso se propone hacer como Maven, y mover todas estas opciones de configuración a un fichero, que sería leído al ejecutar la herramienta.
- **Integración con IDE.** Hacer que la herramienta sea ejecutable desde un IDE la hace más atractiva y fácil de usar, sobre todo si se implementa el punto anterior.

También cabe destacar un producto completamente diferente: una herramienta que, usando inteligencia artificial, aprenda el estilo de codificación de un programador, y con esa información transforme el código que escribe de forma que quede conforme a los estándares de la empresa en la que trabaje. También funcionaría de forma inversa, cogería el código desde el repositorio de la empresa y lo reescribiría de forma que el programador se encuentre a gusto.

Anexo I: Repositorio

Con el objetivo de facilitar la continuación del trabajo iniciado por este proyecto, se puede consultar el siguiente enlace:

https://github.com/Morgloz/TFG

Contiene todos los archivos creados durante el desarrollo de este TFG además de las instrucciones de instalación del entorno de desarrollo.

Bibliografía

[1] Fernández Rodríguez Adrián, "Análisis, Diseño e Implementación de un Prototipo de Transpilador para Interceptar Código" (TFG, Universidad Carlos III de Madrid).

[2] "IntelliJ IDEA," JetBrains, accesed Spetember 25, 2018, https://www.jetbrains.com/idea/

[3] "Eclipse," The Eclipse Foundation, accesed Spetember 25, 2018, http://www.eclipse.org/

[4] "100+ Server Monitoring & Application Performance Monitoring Solutions, " Hayden James, last modified August 2, 2018, https://haydenjames.io/50-top-server-monitoring-application-performance-monitoring-apm-solutions/

[5] "TIOBE Index," TIOBE, last modified September 1, 2018, https://www.tiobe.com/tiobe-index/

[6] "Comparison of parser generators," Wikipedia, last modified September 24, 2018, https://en.wikipedia.org/wiki/Comparison_of_parser_generators

[7] "Bison 1.25 - Introduction," The Lex & Yacc Page, accessed July 15, 2018 , http://dinosaur.compilertools.net/bison/bison_1.html

[8] "Flex - a scanner generator," The Lex & Yacc Page, accessed July 15, 2018, http://dinosaur.compilertools.net/flex/flex_4.html

[9] "ANTLR," ANTLR, accessed July 15, 2018, http://www.antlr.org/index.html

[10] "Clang," Clang Project, accessed August 9, 2018, https://clang.llvm.org/index.html

[11] "Grammars v4," ANTLR, accessed July 15, 2018, https://github.com/antlr/grammars-v4

[12] "LLVM Compiler," LLVM Project, last accessed August 5, 2018, http://www.llvm.org/

[13] "Choosing the right interface for your Application," Clang 8 Documentation, last accessed August 7, 2018, http://clang.llvm.org/docs/Tooling.html

[14] "Documento DOUE-L-2016-80807," BOE, last modified May 4, 2016 https://www.boe.es/buscar/doc.php?id=DOUE-L-2016-80807

[15] "Documento consolidado BOE-A-1996-8930," BOE, last modified April 22, 2016, https://www.boe.es/buscar/act.php?id=BOE-A-1996-8930

[16] "Clang features," Clang Project, accessed August 9, 2018, https://clang.llvm.org/features.html

[17] "The BSD-3-Clause License," Open source initiative, accessed August 9, 2018, https://opensource.org/licenses/BSD-3-Clause

[18] "LibTooling Tutorial," Clang 8 Documentation, accessed August 9, 2018, https://clang.llvm.org/docs/LibASTMatchersTutorial.html

[19]